Auto-configuration of Savants in a Complex, Variable Network

by

Joseph Hon Yu
S.B. Mathematics

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2005

Author .....
Department of Electrical Engineering and Computer Science
May 19, 2005

Certified by .....
Daniel W. Engels
Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Auto-configuration of Savants in a Complex, Variable Network

by

Joseph Hon Yu

S.B. Mathematics

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2005, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I present a system design that enables Savants to automatically configure both their network settings and their required application programs when connected to an intelligent data management and application system. Savants are intelligent routers in a large network used to manage the data and events related to communications with electronic identification tags [10]. The ubiquitous nature of the identification tags and the access points that communicate with them requires an information and management system that is equally ubiquitous and able to deal with huge volumes of data. The Savant systems were designed to be such a ubiquitous information and management system. Deploying any ubiquitous system is difficult, and automation is required to streamline its deployment and improve system management, reliability, and performance. My solution to this auto-configuration problem uses NETCONF as a standard language and protocol for configuration communication among Savants. It also uses the Content-Addressable Network (CAN) as a discovery service to help Savants locate configuration information, since a new Savant may not have information about the network structure. With these tools, new Savants can configure themselves automatically with the help of other Savants. Specifically, they can configure their network settings, download and set up software, and integrate with network distributed applications. Future work could expand upon my project by studying an implementation, making provisions for resource-limited Savants, or improving security.

Thesis Supervisor: Daniel W. Engels
Title: Research Scientist

Acknowledgements

I would like to thank Daniel Engels for providing great amounts of guidance and insight throughout the development of my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Radio frequency identification (RFID) is a method of transmitting and storing data through a wireless connection [15]. Already used in various applications such as electronic toll collection and proximity-based keycard building entry, RFID is becoming more popular everyday. Many other identification systems may also soon be replaced by RFID, such as UPC barcodes. Widespread use of RFID will generate huge volumes of data, which will need to be transmitted, stored, and managed within the connected information systems.

Savants are intelligent routers that are capable of handling the large volumes of data generated by RFID. If they are to be used in the many applications of RFID, there will be many Savants to set up, configure, and maintain. To lessen the burden of this resource-consuming task, I have designed a system that will allow new Savants to automatically configure themselves by communicating with other Savants in the network.

My solution uses NETCONF [11] as a standard language and protocol for configuration communication among Savants. It also uses the Content-Addressable Network (CAN) [12] as a discovery service to help Savants locate configuration information, since a new Savant may not have information about the network contents. When a new Savant arrives, it uses CAN to locate its configuration data, which it will receive using the NETCONF language. It will then configure itself, downloading software as necessary, and join the functionality of

the Savant network.

1.1 The Savant

Savants are intelligent routers that are combined to create a large distributed network to manage data and events obtained from identification tags and perform applications based on the data and events [10]. Because of the ubiquitous deployment of identification tags, from factories to warehouses to stores to consumers' homes and beyond, the Savant network can be extremely large and complex. In addition, the internal workings of the network can be potentially confidential, requiring appropriate security, or complex, making the data difficult to understand. For all of these reasons, Savant system configuration can be rather time-consuming and error-prone when performed manually, so automating Savant deployment can yield valuable benefits.

Bringing up a Savant network from scratch involves the configuration of many machines, which can pose a serious resource barrier for implementation. Furthermore, the maintenance of such a network can involve the addition or removal of many Savants, potentially on a frequent basis, which also demands resources for configuration. My solution to reduce these costs is a system that automates the configuration of new Savants in the network.

The goal of Savant auto-configuration is to be able to place an unconfigured unit anywhere in the network such that it will communicate with other networked devices to locate and configure itself. Finding nearby Savants, downloading software, and communicating with data centers are all parts of this process. The languages, protocols, and services used are important for the architecture and performance of the system.

1.2 Auto-configuration Requirements

There are two main aspects of Savant configuration: network configuration and system configuration. The former is necessary for connectivity to perform its data routing functionality

as well as to gather additional configuration data from other Savants. System configuration is setting up the software of a Savant so that it can contribute to the network's ID tag management functionality.

A discovery service allows a device to obtain information from the network without knowing which machines store what. When a new Savant first arrives, it knows nothing about its surroundings, so a discovery service is a good way for the machine to obtain the configuration data it needs. Many possible discovery service implementations have already been researched, implemented, and documented, with various strengths and weaknesses. Scalability, reliability, and security are important factors for choosing a discovery service for Savant auto-configuration.

The method of communication between Savants is also a key design decision. The system needs a language and a protocol that can transmit the aforementioned network and system configuration data. Since this information can be both complex and confidential, flexibility and security are required.

Security is the one critical requirement that spans across all aspects of the auto-configuration system. In the event of a compromise, the network may cease to function correctly and confidential information may be exposed. Authentication of Savants and encryption of communicated data are two important aspects of security that must be implemented.

1.3 Dissertation Overview

Since RFID may soon be ubiquitous, there may be very large numbers of Savants that need to be set up and configured. Automating this configuration can heavily reduce the resources needed to set up Savant networks.

The first problem to solve is the language that will be used to communicate configuration data. It should be flexible, supporting both network configuration and system configuration communication. NETCONF, the language used in my system, is secure and flexible, supporting any kind of configuration. In Chapter 3, I discuss in more detail the requirements

for the language and why I chose NETCONF.

The next step is a discovery service to allow new Savants to locate and acquire configuration data. Reliability is an important factor because a failure of the discovery service renders the entire auto-configuration system useless, necessitating the tedious manual configuration which was to be avoided in the first place. In addition, the discovery service should be compatible with as many different types of Savant hardware as possible. Security is also very important, since the wireless aspect of RFID can introduce many risks to the Savant network. The CAN discovery service allows for maximum compatibility among all different types of Savants, with a simple constant space requirement. I discuss and compare several existing discovery services in Chapter 4, and I analyze ways to implement them in the Savant network in Chapter 5.

These tools can be put together to create a system in which Savants configure themselves automatically. First, the new Savant broadcasts to find the local leader of its subnet, which aids it in determining its network configuration. Once the Savant is configured to communicate with the network, it uses the CAN discovery service to determine its system configuration. There are various ways to implement the discovery service in the Savant network, depending on the details of implementation. Also, security measures such as encryption, authentication, and redundancy are taken to protect against intruder attacks. The entire auto-configuration process is described in detail in Chapter 7.

Chapter 2

System Overview

This chapter gives an overview of the auto-configuration system. First, there are the choices of language and discovery service, which are the basic tools for finding and communicating configuration data. I also briefly discuss other concerns that affect the design and implementation of the system. Finally, the solution incorporates all of these, as shown in Figure 2-1, creating a full system design for auto-configuration.

2.1 Language and Communication Protocol

By standardizing configuration communication, all different types of Savants can have compatibility with each other, regardless of what function they serve or where they are. The use of a single language and communication protocol is an effective standard. The language needs to be able to convey both network and system configuration. The former includes network protocol (TCP/IP possibly) configuration as well as information relating to the network organization or topology. System configuration includes anything necessary to run the Savant network applications, such as what software to download and how to use it.

After analyzing the benefits of two languages, I have determined that NETCONF is suitable for the auto-configuration system because it is both flexible and secure.

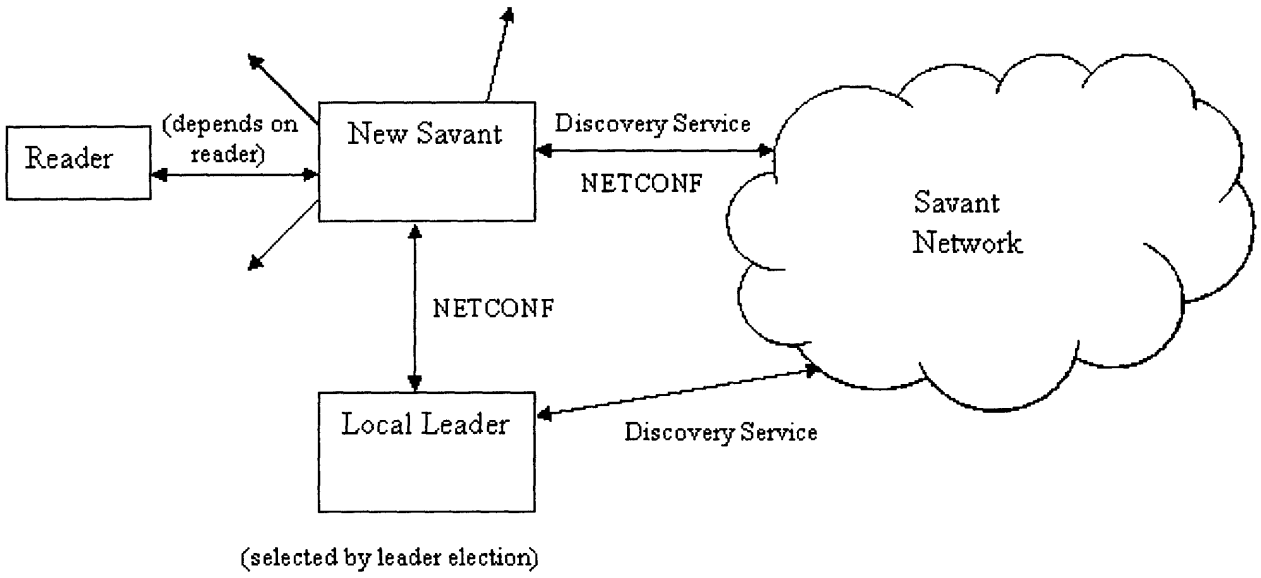

Figure 2-1: A diagram of the Savant network, showing the interaction of devices and the discrete components used. The arrows between devices are labeled with the components used to aid communication.

2.2 Discovery Service

When a new Savant is placed into the network, it needs a way to locate appropriate resources, since it may not have specific information about the structure of the network. A discovery service is a distributed application that provides a method of accessing resources on a network without knowing their exact location. This tool can ease Savant configuration by providing a method for information retrieval and manipulation. The discovery service can also dynamically adjust the way resources are located and accessed when the network changes either intentionally or unintentionally.

I have researched and documented the details of three discovery services. Because CAN has a constant space requirement, it is the most compatible of the three for Savants. To maximize the usability of the auto-configuration system with various types of Savant networks, I have chosen CAN for my solution.

2.3 Discovery Services and the Savant Network

After choosing the language and discovery service, there is still the question of how to use them in the Savant network. There are a few different schemes, with important trade-offs between them. Some are more complex, but can be more efficient and more secure depending on the specific characteristics of the implemented Savant network. Because the benefits vary depending on the implementation, no clear choice arises in the design process.

2.4 Other Concerns

After choosing the language and discovery service, a number of other factors are also important for auto-configuration. A new Savant needs to have a way to download software. The must also be specified formats for configuration data. The Savant network as a whole must also be managed to keep track of additions and removals.

2.5 Solution

All of these factors can be put together to form a system that enables a new Savant can automatically configure itself. First, another Savant helps it to configure its network settings. Afterwards, it uses the discovery service determine its function. Following that, it downloads, installs, and configures its software. Finally, it incorporates itself into the Savant network functionality. Security measures, such as encryption, authentication, and redundancy, are taken to reduce the risk of attacks.

Chapter 3

Language and Communication Protocol

Communication of configuration data should be standardized such that Savants do not need to use different methods to communicate with one Savant or another. Also, it would be beneficial to have one set of auto-configuration code that works regardless of what network the new Savant is placed into. One language and communication protocol can serve as this standard.

There are two types of configuration data to be standardized: network configuration and system configuration data. When evaluating candidates, it is important to consider which of these two types of data they can handle. In this section I evaluate and compare two candidates for this protocol, DHCP and NETCONF. While DHCP can only be used for network configuration data, NETCONF can be used for both types [5] [11].

3.1 Dynamic Host Configuration Protocol (DHCP)

Dynamic Host Configuration Protocol (DHCP) is a configuration protocol used to automate network configuration. Many networks currently use DHCP for TCP/IP configuration needs.

A new computer in the network acts as a DHCP client, while an established machine acts as a DHCP server. They undergo an interchange of requests, responses, and acknowledgments after which the client has configured itself with an IP address and the server has updated its database to prevent IP collisions. DHCP is a flexible protocol that allows different methods of IP address assignment and management [5].

DHCP is a widely used system that effectively reduces the administration overhead of using TCP/IP. It works best, however, in hierarchical network topologies where individual DHCP servers can manage subnetworks with centralized authority. DHCP provides a minimum level of security. Savants will require a higher level of security than that provided by DHCP [5].

3.2 NETCONF

The NETCONF configuration protocol specifies a way to manage configuration data for network devices. Using Remote Procedure Calls (RPC) encoded in eXtensible Markup Language (XML), it provides a flexible but fully specified interface for Savant communication. It supports several operations, including the retrieval and manipulation of both network and system configuration [11].

The semantics of NETCONF can be partitioned into four layers: application protocol, RPC, operations, and content (shown in Figure 3-1). The application protocol sends and receives data through the network connection. RPC is the framing mechanism for NETCONF communication, taking as input an RPC and encoding it in a way that can be communicated through the application protocol. The operations layer defines a set of functionality, translating an NETCONF request into RPC. Finally, the content layer uses NETCONF functionality for some greater purpose, such as configuring Savants to integrate into the network and its running applications [11].

NETCONF has a set of basic requirements for its application protocol. First, the protocol must be able to maintain a persistent connection, including sending data correctly and in

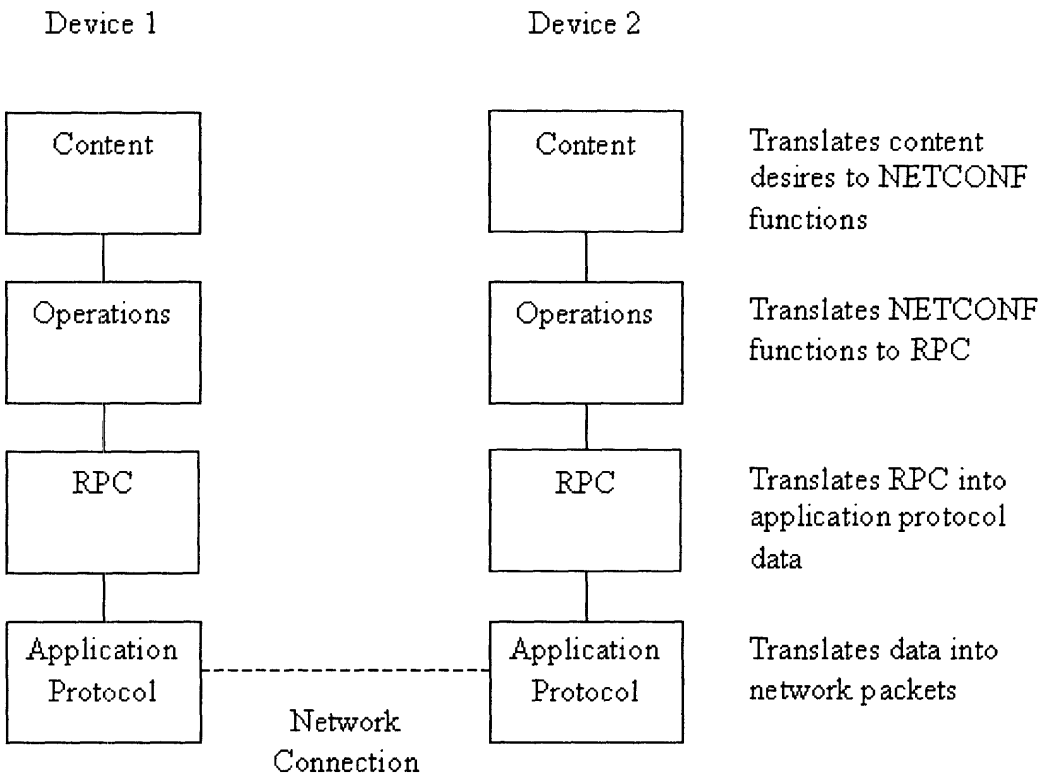

Figure 3-1: The layers of NETCONF.

order. NETCONF leaves security considerations a responsibility of the underlying protocol. This includes authentication, data integrity, and privacy. Secure Shell (SSH) is a connection protocol that satisfies the application protocol requirements, including security [11].

Because of this application protocol requirement, using NETCONF without a basic network protocol (such as TCP/IP) could be difficult. One possibility for this initial setup is to run some application protocol on top of MAC address packets, so that the entire network configuration sequence can use NETCONF [11].

3.3 Comparison

DHCP is simple and proven for existing computer networks, but the Savant network may differ from the common system layout of many networks. The common usage of DHCP currently is a single machine managing IP addresses and serving requests for a group of machines. The Savant network may lack a structure that incorporates this centralized model well, so using DHCP may prove difficult. NETCONF, on the other hand, is not very prevalent, but its emphasis on flexibility suits the Savant network well. In addition, NETCONF can run more securely than DHCP with a secure application protocol. For these reasons, my choice is NETCONF for the configuration language.

Another advantage of choosing NETCONF is that it can be used to communicate both network configuration and system configuration data. Using the same language and protocol for all types of configuration will unify the design and make it easier to build generalized tools during implementation.

Choosing NETCONF, however, does not mean that DHCP is no longer a consideration for implementation. Since DHCP is such a widespread standard today, many devices may be easiest to set up in a DHCP environment. It would be possible to implement a mixed environment in which devices use DHCP to conduct basic network configuration, but use NETCONF for advanced network configuration and system configuration.

3.4 Security Concerns

NETCONF is secure to the extent that the means of passing its messages are secure. These messages will be passed as part of discovery service interactions, when Savants will obtain configuration data. For this reason, I will discuss security in more detail when analyzing discovery services.

Chapter 4

Discovery Service

When a new Savant first enters the network, it may not have any information about the layout of the network or where it is in the network. A discovery service is a system that allows access to network resources without knowing their exact location. This can help a Savant locate configuration information when it first arrives.

Numerous discovery services exist, some for special usage and some for general, each with their own advantages and disadvantages. To begin, some discovery services provide ways to request and transmit pure data, while some allow searching for various services and applications other machines may be running. Either of these could be useful for the problem at hand.

A data discovery service is the most straightforward, since when a new Savant arrives it must simply acquire some data to be able to configure itself. This kind of system, however, may consume a large amount of network bandwidth, since data requests are not directed towards any individual machine. This is a more distributed model and can be very robust, even in the face of multiple failures.

A discovery service that searches for services provided by the machines creates a familiar client-server environment to conduct configuration. A stable, secure connection can be formed to transmit data. Also, finding a download server is straightforward in this system,

as it can simply advertise itself as an available file server. This is a more centralized model, which is simple and familiar but can suffer badly if an important machine fails.

Overall, the distributed model is more robust with respect to network changes and node failures. Also, it is easier for a client to send out one request and receive data in return than to find a service, connect to the server, and negotiate a data transfer. I have thus chosen to study three data discovery services in detail. Of these, CAN is used in my solution.

4.1 Previous Work

The Globe system maps object identifiers to locations, arranged as a large hierarchy [1]. Although it provides fast performance and exploits network locality well, it cannot be used for the Savants' possibly unstructured network topology.

The Freenet peer-to-peer storage system supports anonymous storage and retrieval of distributed data [16]. No machine or group of machines is responsible for storing and/or maintaining particular data files. Data retrieval searches for the closest copy of a particular item; data is stored in locations of great interest and likewise deleted from those with no interest. To provide anonymity, however, Freenet sacrifices performance.

Czerwinsky discusses a design for a secure Service Discovery Service (SDS), which allows clients to search for services provided by other machines on the network [4]. The definition of a service with respect to this design report is any application that performs an action on behalf of a client request. The Ninja SDS documented prioritizes robustness, scalability, and security. There are periodic multicast announcements to declare which services are available. Presence of such announcements, or lack thereof, maintains caches on SDS servers of available services. There is no separate procedure for recovery failure, but rather one simple mechanism is used to do everything. The SDS servers form a hierarchical structure with dynamically distributed load to provide scalability when the number of services increases in one region of the network or another. Finally, encryption and authentication are key components of the SDS, which allows a certain amount of trust to be assumed in its design

and implementation.

Sensor networks exhibit some similarities to the Savant network, including running distributed applications robustly, even when no individual component may be considered reliable. Although the exact algorithms and design tactics used for these networks may not be directly applicable, certain concepts or lessons learned by sensor network design projects may be valuable in the solution to the auto-configuration problem [7] [9].

4.2 Pastry

Pastry is a system that provides basic functionality which can be used to build a variety of information sharing and distribution applications. Each node in a Pastry network has a node ID. Given a message with a numeric destination key, Pastry will route the message to the node whose node ID is numerically closest to the key. Pastry can also deliver more loosely, ensuring that the message reaches one of the k nodes numerically closest to the key, given some integer k. Only one of these k nodes must be live to ensure the delivery. In terms of a network locality metric, the closest of these nodes to the sender will receive the message [13].

One application that uses the Pastry substrate is PAST, a distributed storage utility. Given a file, a key is calculated as a hash of the file's basic information, such as its name, owner, and size. The file is then stored redundantly on k nodes with node ID closest to the key. Then, any node can find this file using the Pastry substrate [13].

In a Pastry network, each node maintains some information about nodes that are close to it, with respect to both node ID and network proximity. There is a tradeoff between the amount of information stored per node and the expected number of hops required to deliver a message. A configuration parameter can be set to determine how much information is stored [13].

In a network with N machines, each node stores information about $O(\log N)$ other nodes. On average, $O(\log N)$ hops are required to pass a message from one node to another. An

experimental implementation of Pastry showed that the average path length ranged from 2.5 for 1000 nodes to 4 for 100,000 nodes. In a network with 20,000 nodes, the average path length was 3.5 hops [13].

4.3 The Chord Project

The Chord Project is a peer-to-peer discovery service that emphasizes simplicity, provable correctness, and provable performance. To accomplish this, it uses a consistent hash function to distribute data evenly among the devices in the network with high probability. In addition, each machine only needs to hold a small amount of routing information about other devices, so that the costs of running Chord do not increase much as the network grows. Scalability is also emphasized with regards to the costs of locating information and maintaining the state of the discovery service when machines join and leave. For simplicity, Chord uses a flat key space to which an application can map data in whatever manner is convenient [8].

In a network with N nodes, each one maintains information about $O(\log N)$ other nodes. The expected number of hops to resolve a lookup is $O(\log N)$. In experimental simulation, the average path length ranged from 2 for 10 nodes to 7 for 20,000 nodes [8].

4.4 The Content-Addressable Network

Ratnasamy, et al. developed a Content-Addressable Network (CAN) as a way to store information in a distributed way over a network of systems. The authors observed that hash tables are an effective storage mechanism on a single machine, and CAN is an extension to network storage. A hash function maps to a Cartesian coordinate space, which is divided into zones, each of which is owned by a node. Then, when storing or retrieving data, the hash function, operating on some aspect of the data, determines a point in the coordinate space to which the data corresponds. The system that owns the zone containing the point is responsible for the data. Many variations, such as the number of dimensions of the coordinate

space, routing algorithm, and hash function, can affect the performance of CAN [12].

Given a network of N nodes, and d, the number of dimensions in the coordinate space: Each node stores information about $2d$ neighbors. The average routing path length is $O(N^{1/d})$. In simulations with $d = 10$, the average path length was 4.56 hops with 2^{14} nodes, and it was 5 hops with 2^{18} nodes [12].

4.5 Comparison

The asymptotic performance estimates in Table 4.1 show a tradeoff between space and time. Although Chord and Pastry are faster in search time than CAN, they require more storage on each node. In fact, CAN's storage requirements are constant, so they should be simple to implement.

For Savant auto-configuration, the speed of the configuration is not very important, as long as it is reasonable. Reliability and compatibility are, on the other hand, key factors to its operation. Since Savants may sometimes be very small machines with little available storage, a constant space requirement can simplify implementation. A system running the discovery service will always be able to do so, no matter how large the network becomes.

On the other hand, if the network is very large and its components are not too small, it may be worth it to allow for a higher storage requirement in exchange for faster searches. Specifically, for very large values of N, $O(N^{1/d})$ can be much larger than $O(\log N)$.

Many Savant networks will be small in size, on the order of 10 or 20. In a much larger network with 20,000 nodes, all three of the studied discovery services perform well, averaging less than 10 hops per lookup. Also, as there are more and more opportunities for RFID and Savants to be implemented, Savants may become smaller and smaller. To maximize the usability of the auto-configuration system, the discovery service should be compatible with devices as small as possible. To this end, it is worth it to have slower lookups in exchange for a constant space requirement. For this reason, CAN is my discovery service of choice for this system.

Table 4.1: Performance of discovery services. N is the number of nodes and d is the number of dimensions, a parameter for CAN. The actual numbers are based on simulations for Chord and CAN, and on an actual implementation for Pastry.

Discovery Service	Average Lookup Path Length (hops)		Space Requirement
Number of Nodes	20,000	N	N
Pastry	3.5	$O(\log N)$	$O(\log N)$
Chord	7	$O(\log N)$	$O(\log N)$
CAN	5	$O(N^{1/d})$	$O(2d)$

4.6 Security Concerns

The discovery services presented here are open to a number of malicious attacks. Because of the distributed nature of the services, it is more difficult to secure the network against these attacks than in networks without distributed applications.

One attack that caters to the functionality of the discovery services is mimicry. By pretending to be a client of the discovery service, an unauthorized machine can gather sensitive information from the discovery service simply by placing lookup requests. The same machine could also claim to serve data, responding to lookup requests with incorrect information.

The most direct way to protect against this kind of attack is to require some form of authentication for all Savants [6]. Attaching cryptographic signatures to all messages is a common method.

Another common attack is the man-in-the-middle attack, which entails intercepting, reading, and modifying network transmissions. It can often be difficult to detect this kind of attack, since it can be invisible to the network.

A common countermeasure is to encrypt all messages such that they cannot be viewed or modified without the right key [6]. Unfortunately, since lookup requests are not sent to a specific target, all Savants in the network need access to the private key that decrypts these requests. This lowers the effectiveness of the countermeasure because the compromise of a single Savant can cause it to fail.

Other times an attacker will attempt to block access to a resource. This is often known as a denial-of-service attack. This can be done by simply disconnecting physical components, but even without physical access it can be done by flooding network traffic.

One way to handle denial-of-service attacks is to detect the attacker and use it to prevent further attack. For example, in the case of the network flood attack, by dropping all incoming packets that come from the attacker, the server can function normally. It is not always easy, however, to detect the source of the attacks.

Another way to prevent against denial-of-service attacks is to have redundancy, so that if one machine is disabled, another one can provide resources in its place. This is convenient for the auto-configuration system, since many discovery services already have provisions for redundant storage across multiple nodes.

Chapter 5

Discovery Services and the Savant Network

There are a variety of ways to use a given discovery service in the Savant network. The information a new Savant seeks can be very close to it or very far. The lookup should not go very far away if the information is close. Also, if the information is far, the lookup path length should be minimized if possible.

The following section describes details of the Savant network that are important to auto-configuration implementation. Afterwards, I discuss the different implementation possibilities given these details.

5.1 Savant Network Details

The topology of the Savant network affects the performance of the discovery service, so it is a major factor in auto-configuration design and implementation. Some discovery services operate under the assumptions that no machine is reliably connected to the network and there is always uncertainty when communicating. Other discovery services, by contrast, conduct benchmarks and testing in controlled, organized, and structured environments.

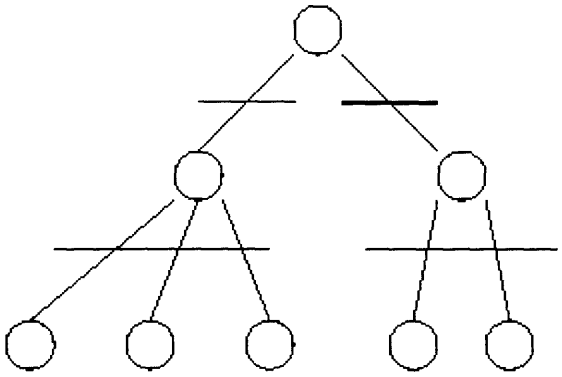

Figure 5-1: An example of a tree topology. The top node is connected to multiple down subnets, while each of the second-tier nodes is connected to only one down subnet.

Generally, a Savant can have one "up" subnet and one "down" subnet, although in some topologies the Savant will connect to multiple subnets on either side [10]. The two basic topologies exhibiting this are the tree topology and the layered topology. In the tree topology, each computer connects to only one subnet up and at least one subnet down, as in Figure 5-1. In some tree networks, all of the computers directly below another computer are on a single subnet, but in some networks this restriction doesn't exist. In the layered topology (Figure 5-2), there are multiple layers, each which has one or more subnets. Each computer exists between two adjacent layers, and can connect to one or more subnets in each of those layers. The layers can be in the ladder formation, where one of the layers is at the top, one of layers is at the bottom, and the rest are ordered in between. Another possibility is the ring formation, which is similar to the ladder formation but the top and bottom are connected. The ring adds a level of redundancy, since if one layer fails, the network will become a ladder formation and will maintain full connectivity.

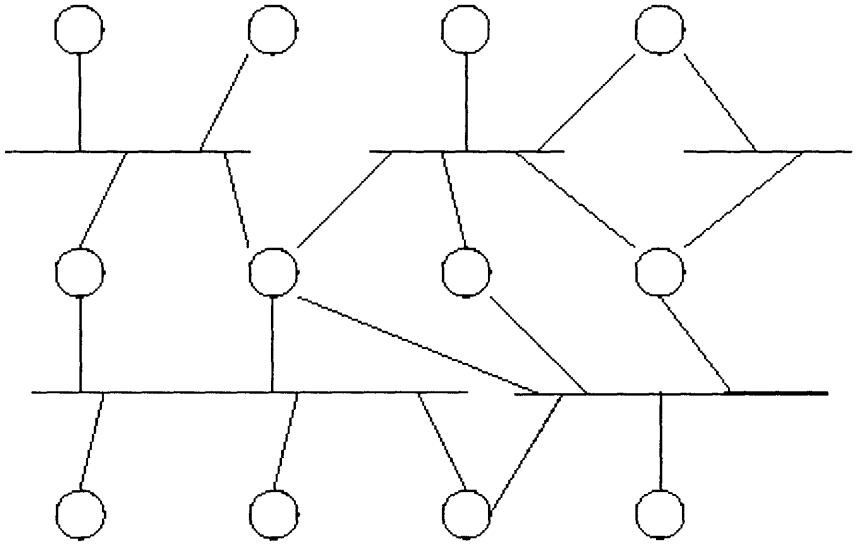

Figure 5-2: An example of a layered topology. Each layer in this network has multiple subnets.

5.1.1 Local Leader Selection

In either of the two aforementioned topologies, the Savant network is divided into subnets. To manage the discovery service traffic going through these groups, a leader may be elected for each. The key requirement is that there is always one and only one leader; the election algorithm must dynamically check and correct this.

In a tree topology, each Savant has several other Savants connected below it on one interface, but only one above it on a different interface. In this case, a typical behavior would be for each Savant to be the leader on its "down" interface. Meanwhile, on the "up" interface, the Savant would assume that another machine is the leader.

One example of leader election in previous networking studies is token regeneration in token rings [3]. The token is passed from one device to another, and only the device that has the token is allowed to transmit data. This prevents collisions normally caused by two machines sending data at the same time. The problem is that, if machine failure or loss

of connectivity causes the token to disappear, communication will come to a halt. Token regeneration algorithms recover token ring networks from such a state.

Salonidis, et al. create a simple leader election algorithm for Bluetooth devices [14]. Each device starts by setting their VOTES variable to 1. Then, the devices randomly connect to each other in one-on-one confrontation. The device with a higher VOTES variable wins each confrontation. If it is a tie, the device with a higher Bluetooth address wins. The loser sends to the winner the information of all the devices it is one thus far. Eventually, one device has all the information and is the leader.

Brunekreef discusses the development of a leader election algorithm for dynamic systems [2]. The focus of the design is maintaining a leader when the network elements can change continuously. This would be a good algorithm for networks where Savants fail or lose connectivity often.

The Bluetooth implementation is simple and works. It can be adapted to the Savant network as follows. Each Savant with a positive VOTES variable periodically broadcasts its subnet to establish its presence. If more than one Savant broadcasts in this way, they engage each other in one-on-one confrontation. If only one Savant remains broadcasting for a timeout period, it is declared the leader. Also, if no Savant broadcasts for a timeout period, all Savants set their VOTES variable to 1 and the election process begins again. The confrontations work as in the Bluetooth implementation, except the MAC address is used as the tiebreaker instead of the Bluetooth address.

If it is determined that the network elements can change very often, then the Brunekreef implementation could be better. I leave it to the implementer to analyze the potential benefits given his specific network.

5.2 Ignore the Subnets

The most direct implementation is to simply run the discovery service on the whole network. This would require that each Savant have network connectivity with each other Savant, with

TCP/IP gateways or the like connecting subnets. It should be easy to make performance estimates, since many discovery service papers use a single network as the main test substrate.

If a subnet has on average N nodes, and there are M subnets connected in a ring formation, the entire network will have MN nodes. Therefore, a lookup will take $O(\log(MN))$ hops using Chord or Pastry, and $O((MN)^{1/d})$ hops using CAN.

5.3 Two Levels of Discovery Services

Run one copy of the discovery service on each subnet. This will serve all information requests that can be satisfied without leaving the subnet. Then, the subnet local leaders will run another copy of the discovery service among themselves. If information cannot be found within a subnet, the leader will use this second-level discovery service to find information. Likewise, when a leader needs to provide information from its subnet to another leader, it will use its first-level discovery service to obtain it. Figure 5-3 illustrates the interaction between Savants in this scheme.

If a subnet has on average N nodes, and there are M subnets connected in a ring formation, then a lookup will typically require one first-level lookup, one second-level lookup, and finally another first-level lookup. This would take $O(\log M + 2\log N)$ hops using Chord or Pastry, and $O(M^{1/d} + 2N^{1/d})$ hops using CAN.

5.4 Discovery Service Forwarding through Gateways

Another solution is to have the gateways between the subnets forward discovery service requests in an intelligent manner. As in the two-level scheme, each subnet runs one copy of the discovery service. When a gateway node receives a lookup request from a subnet that does not hold the information, the gateway forwards the request to the other subnet that it is connected to. To prevent endless cycles of forwarding, each gateway needs to keep track of what requests have already been forwarded. In Figure 5-4, a request is forwarded in a

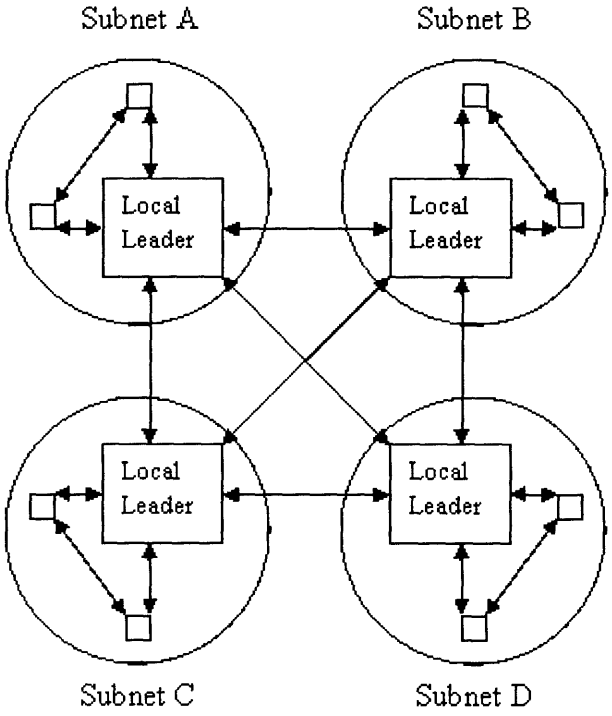

Figure 5-3: There are two levels of discovery services in this scheme. Each subnet has a copy of the discovery service running, and another copy connects the local leaders of all the subnets.

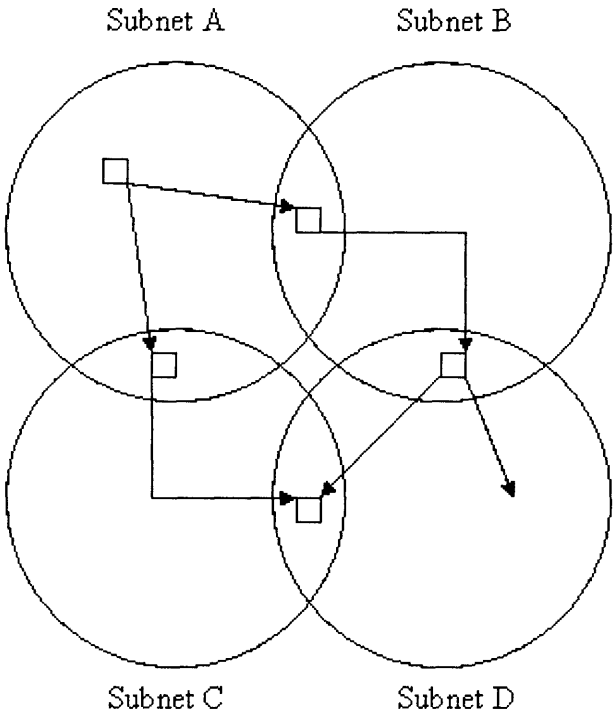

Figure 5-4: A discovery service request makes its way from subnet A to subnet D. The gateways, which each lie in 2 subnets, forward the request to subnets B and C. Afterwards, another gateway forwards the request to subnet D from subnet B. Since the request has already been forwarded to subnets C and D, the C-D gateway will not forward this request.

simple network with four subnets.

The scheme does not require a local leader to be established for each subnet like the two-level scheme does. Instead of relying on the election algorithm, it relies on the gateways being up. Since a gateway failure prevents any traffic from flowing, it is no further loss for the discovery service to fail. Overall, forwarding through gateways is more robust than using local leaders for the discovery service scheme.

If a subnet has on average N nodes, and there are M subnets connected in a ring formation, then a lookup will on average go through $M/4$ subnets. This will take $O(M \log N)$ hops using Chord or Pastry, and $O(MN^{1/d})$ hops using CAN.

Table 5.1: Asymptotic performance of discovery service schemes in the Savant network using the CAN discovery service. M is the number of subnets and N is the number of nodes in each subnet. d is the number of dimensions, a parameter for CAN.

Scheme	Lookup Time
Ignore Subnets	$O((MN)^{1/d})$
Two Levels	$O(M^{1/d} + 2N^{1/d})$
Gateway Forwarding	$O(MN^{1/d})$

5.5 Comparison

Examining the data in Table 5.1, when using CAN as the discovery service, as the size of the subnets and the number of subnets increase, the two-level scheme performs best, while the gateway forwarding scheme is the slowest.

Several factors affect the advantages and disadvantages of these different implementation schemes. The proximity of information is significant. In my performance analysis, I assumed that data was distributed randomly throughout the network. If many searches can be resolved within the subnet, it saves time and bandwidth to check before searching other subnets, so the two complex schemes perform well. On the other hand, if many searches must go outside the subnet, it is wasted overhead to check every time.

With regards to implementation, the first approach is simplest, as it only has one copy of the discovery service and Savants do not need extra code to keep track of what specific data other Savants provide. For small Savant networks that need to be implemented quickly, this approach is suitable. For larger networks, further analysis could determine what benefits could be gained for the specific characteristics of the Savant network being implemented.

5.6 Security Concerns

The two multi-level schemes have security advantages, since searches do not go out of a subnet unless they have to. Therefore, there are limits on the incorrect information an

imitating attacker can provide to a Savant that is not in its subnet. Man-in-the-middle attackers are similarly limited. On the other hand, this limitation can make detecting attackers more difficult. If, for example, an entire subnet is compromised, and most of its configuration information is self-contained, Savants outside the subnet may not be able to detect its compromise, because they do not interact with it for configuration purposes.

Another serious concern for the two more complex schemes is that they are partially centralized. The local leaders and gateways, respectively, are important for the functionality of these two schemes, so they are vulnerable attack points. These nodes must therefore be guarded more closely than the other Savants.

Chapter 6

Other Concerns

6.1 Determination of Function

The Savant needs to be assigned a function based on its location information. Since different instances of the Savant network can have vastly differing topologies and functionality, the mapping of location to function will be far from constant. The methods for creating, maintaining, and accessing this mapping, however, can be designed to be used in all cases, flexible to the different structure and function of each instance.

The most important invariant is that the storage mechanism for this mapping must be accessible via the discovery service. Given this, a new Savant can determine its function after connecting to the network and use that to configure its running applications.

6.2 Acquisition of Necessary Software

Since the function of a Savant is initially unknown, and software may be updated frequently regardless, the new machine will need to acquire its necessary software from the Savant network. It will need to determine what to download and where to find it. After downloading, it will need to install and load it without human intervention.

A new Savant needs to download software based on its function; thus the original authority which assigned the function will also send information for software acquisition. Then the Savant will look for a local copy of the software and download it, which will again be aided by the use of a discovery service.

At this point, the Savant is faced with the possibly complex task of loading the software onto itself. This could involve the replacement of most of the original software, since the available disk space could be low on a small machine. Furthermore, the Savant may need to reboot, loading the new application instead of the old. These are important considerations when designing both the initial software and the more specific applications.

6.3 Network Management

The Savant network is large and highly varied, so its management is very complicated. Additionally, it may not have a single master, but rather have a distributed management scheme. As devices enter and leave the network, the overall dataflow will need to continue seamlessly while the running applications adjust for the changes.

A hierarchical structure can aid in managing the complexity of the network, since there will be a certain amount of locality. Subnetworks may also have specific security regimes or sensitive data which would be contained only in that area.

The distribution of information needs to be easily accessible from all points in the network to allow the aforementioned auto-configuration to take place. The management of the network is closely linked to the discovery service used.

The network management will also need to take account for the dataflow in the system. As it is important for Savants to know where each specific data packet needs to go, the management scheme also must keep track so that it can configure new savants and update old ones when the network changes.

Pertinent to all this is the way that information travels through the network. When a Savant needs to communicate with another Savant far away, does it address its IP directly

or make a request of its local leader? The first is simpler and more standard, but less secure. The latter can decrease network bandwidth requirements with intelligent routing, but it requires more processing power.

Chapter 7

Solution

7.1 Overview

My solution involves a sequence of steps necessary to bring a Savant from its initial state to a fully functional state specific to the location in the network where it is connected. First, this requires some bootstrapping to set up network configuration so that it can communicate with other savants. Afterwards, the Savant needs to download software and integrate itself into the running applications of the network. A diagram of the steps is presented in Figure 7-1.

I use several discrete components in my solution, including the language, discovery service, and leader election algorithm. The auto-configuration system has requirements on each of these. I analyzed several different choices and decided to use NETCONF as the language, CAN as the discovery service, and a modified Bluetooth implementation for the leader election algorithm.

A Savant first needs to make initial contact with one or more nearby Savants to help configure its network settings. It would be redundant, however, for every Savant that receives the message to do all the necessary information gathering. To this end, each local group of Savants should have a leader, which can be selected using an election algorithm. Such a

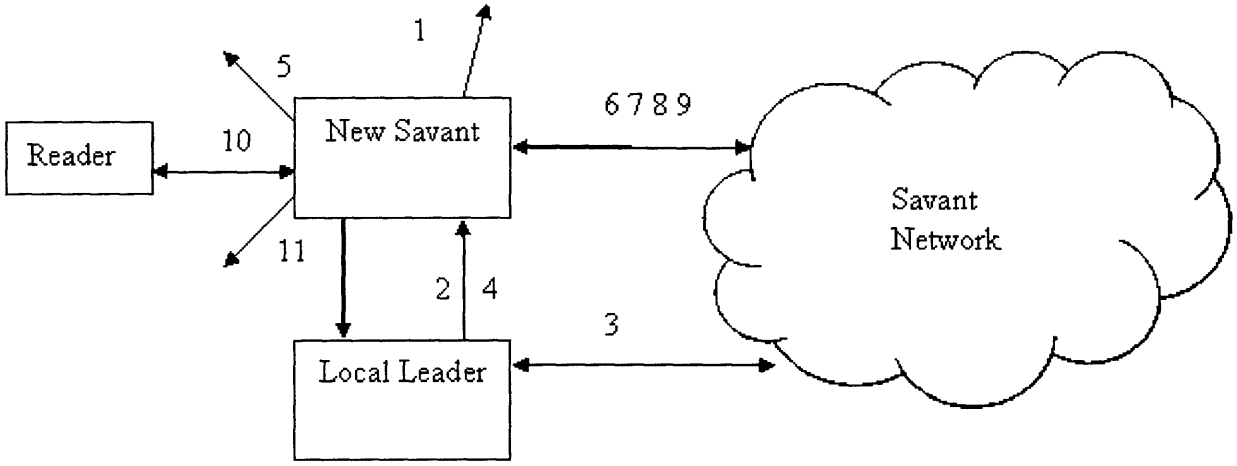

Figure 7-1: The steps for Savant auto-configuration, shown in a diagram of the network. 1) The new Savant broadcasts to announce its arrival. 2) The local leader responds with acknowledgement. 3) The local leader uses the discovery service to acquire the new Savant's network configuration. 4) The local leader sends the network configuration to the new Savant. 5) The new Savant compiles information about itself and its surroundings to prepare for discovery of function. 6) The Savant determines its function using the discovery service. 7) The Savant downloads the software to serve its function. 8) The Savant obtains its software configuration using the discovery service. 9) The Savant updates the discovery service entry listing the Savants in its subnet. 10) The Savant initiates contact with the neighboring ID tag readers. 11) The Savant completes its integration into the network functionality.

system can automatically choose a new leader should there be no leader at any given point in time.

Next, the Savant should determine what its function is within the network. This requires it to compile information about itself. By searching discovery service with this information, the Savant can find its function.

After discovering its function, the Savant can download software. The discovery service lookup in the previous step should produce a list of software programs for the Savant to download. Using the discovery service like a filesharing network, the Savant can download the software.

After installing the software, the Savant needs to configure it. In a similar fashion to determining function, the Savant can search the discovery service for the configuration of each software program.

Finally, the Savant can integrate with the distributed applications of the network. This includes making its presence known, connecting with nearby non-Savant devices, and other steps specific to the functionality of the Savant and the network is a whole.

For all the steps, security is very important. There are several countermeasures that can be taken to protect the auto-configuration system from malicious attacks. Authentication, encryption, and redundancy can protect the system from some of the common types of attacks.

7.2 Discrete Components

There are several components to the auto-configuration system: language, discovery service, and leader election algorithm. In this section, I review previous discussion for the requirements of these components and my choices for each of them.

The language is a standard for configuration communication. By adhering to this standard, Savants can maintain compatibility with each other regardless of their type, location, and age. Also, it allows for code reuse.

The language needs to be flexible, with the ability to transmit network configuration data as well a system configuration data. Regardless of how much the auto-configuration system changes in the future, it would be ideal to continue to use the same configuration language. NETCONF satisfies these requirements. It uses XML encoded messages and can transmit any kind of configuration data.

The purpose of the discovery service is to help Savants obtain information from the network, even though there may not be centralized servers and the network topology may not be known. The basic functionality can be described as a key-value pairing lookup. A machine can use the discovery service to determine the corresponding value for any key.

The discovery service should be reliable and efficient, such that it works for very large networks and the space requirements do not increase much as the network grows. The lookup times should also scale well as the network grows, but this is not as important. The CAN discovery service has a constant space requirement, which makes implementation simpler—no need to worry about running out of storage because of the discovery service. It also increases compatibility for smaller machines, which is important as Savants will become smaller and smaller in the future. For these reasons, I chose CAN as the discovery service for this system.

There are a few different schemes for discovery service implementation in the Savant network. They are all transparent to the new Savant client. Since the benefits of each scheme vary depending on the details of the Savant network, this choice is left to the implementer.

For each subnet there is a local leader chosen by an election algorithm. The requirement is that there is always one and only one leader in each subnet. I use a modification on a Bluetooth leader election algorithm. It uses one-on-one confrontations and broadcasts to maintain the invariant.

7.3 Network Configuration

Before communicating with the Savant network, a new Savant must configure its network settings. This can include TCP/IP settings as well as any other network protocols used in the Savant network.

To determine its network settings, the new Savant needs to communicate with its subnet. Because it has not set up its network configuration, the Savant cannot contact other machines individually, and it must send a broadcast requesting network configuration data, shown as step 1 in Figure 7-1.

The local leader of the subnet is responsible for responding to this request. In some networks, the local leader has all the information it needs to set the new Savant's network configuration. In others, however, it needs to find the information using the discovery service before it can complete the network configuration (step 3 in Figure 7-1). The discovery service lookup may take a significant amount of time, so before performing the lookup the leader should send an acknowledgment of having received the network configuration request (step 2 in Figure 7-1). Once the local leader has all the information needs, it configures the new Savant's network settings using a NETCONF edit-config operation (step 4 in Figure 7-1).

Note that this behavior is not unlike the standard DHCP procedure. If the network configuration is a standard TCP/IP configuration, the new Savant and the local leader could use DHCP to communicate it. The leader could still use the discovery service behind the scenes to determine the configuration data. This alternative could be useful for implementation, because many devices already have compatibility with DHCP.

7.4 Discovery of Function

Now that the new Savant has completed network configuration, it can communicate with the Savant network, and thus it can use the discovery service by itself to determine the rest of its configuration. The first step is to find out what its function will be in the Savant network.

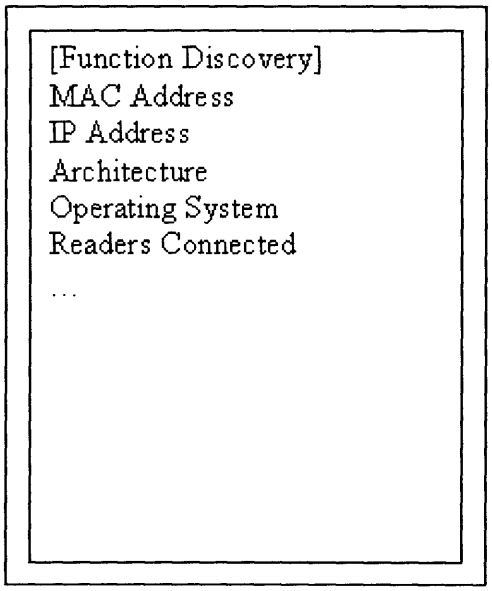

Figure 7-2: An example of a file format for the factors determining Savant function.

The function can include interfacing with ID tag readers, routing data, accessing databases, and more.

The function of a Savant depends on several factors, some of which are network location, physical location, Savant hardware, and connected non-Savant devices. The compilation of these factors should uniquely determine the Savant's function. As such, the new Savant should prepare this data as a first step to function discovery (step 5 in Figure 7-1).

In the implementation of the auto-configuration system, a file format must be designed and agreed upon for the compilation of factors that determine a Savant's function. Because the list of factors can change, the Savant must acquire this file format before determining its function. A discovery service lookup is sufficient for this purpose.

A basic example for this file format is given in Figure 7-2. The results of NETCONF get-config operations can be used for some of the entries, such as IP address and operating system.

Once the Savant has compiled its specific information into this standard file format, it can

use it as a key with which to search the discovery service (step 6 in Figure 7-1). The value that corresponds to this key is a NETCONF edit-config operation that sets the function of the Savant.

7.5 Software Download and Installation

Once a Savant's function is determined, it can take the next step and download the software necessary to perform its duty. As a result of the previous section's NETCONF operation, the Savant should now have a list of software programs to download.

For the purposes of software download, the discovery service can be used like a filesharing application. The name of the software program can be used as the key, and the actual program is the corresponding value. This is illustrated as step 7 of Figure 7-1.

An alternative design could be to have software available from certain servers. In this case, the returned value of the discovery service search would be a list of servers. This could allow the file servers to lie outside the Savant network, to lessen the load on the Savants. Additionally, it might be easier to secure centralized servers, which is important because actual executables are being delivered. On the other hand, the redundancy of the discovery service can help robustness and reliability. Using the discovery service also unifies implementation, since it is being used for many steps of the auto-configuration process.

Each program's package contains the necessary files to install it as well as a standard installer executable that will perform the installation. This way, automated installation is as simple as running this executable for each of the programs.

7.6 Software Configuration

Now that the Savant has installed the necessary software, it needs to configure it. Similarly to the discovery of function step, software configuration depends on many factors. These factors can be different for each software program. As such, there could potentially be a

file format for every program that encapsulates the factors necessary to configure it. For simplicity's sake, in my solution there is only one file format containing the factors for the configuration of all software programs run in the Savant network.

As in the discovery of function, the new Savant must obtain this file format using the discovery service. Afterwards, it can compile the factors to prepare for software configuration. Finally, it searches the discovery service for the configuration information (step 8 of Figure 7-1).

For each program, the discovery service key is the combination of the program name and the formatted file of configuration factors. The value corresponding to this key is the NETCONF edit-config operation to configure the program. After processing all of these operations, the Savant should have configured all of its software.

7.7 Network Integration

The final step is to integrate into the distributed applications of the Savant network. Firstly, the new Savant should make its presence known to the other Savants in its subnet. Similarly, it should determine a list of these Savants. Both of these can be taken care of with a discovery service entry that is simply a list of Savants in the subnet. The new Savant can acquire this list and add itself to the end of the list (step 9 of Figure 7-1).

Next, the Savant should initiate contact with any other devices it is connected to, such as RFID readers (step 10 of Figure 7-1). The Savant should know these devices as a result of compiling the list of factors to discover its function. For each type of device, the Savant can issue a discovery service lookup to determine the sequence of steps necessary to initialize the connection to that device.

Integrating with the distributed applications of the network is very specific to the software running on the Savant and how it relates to the functionality of the network as a whole. For this reason, each program has a command for network integration. Some programs will do nothing, while others will contact other Savants to establish responsibility for certain

aspects of the distributed applications. The Savant must merely cycle through its programs, running the network integration command for each of them. This will complete the Savant's integration into the functionality of the network (step 11 of Figure 7-1).

7.8 Security

Introducing an automated system such as this comes with several new security concerns. Common attacks such as mimicry, man-in-the-middle, and denial-of-service are always a hazard for the Savant network. Furthermore, the auto-configuration system should have measures in place in case a Savant's software becomes compromised, either before or after its integration into the network.

The following is a public/private key system that allows encrypted message passing and authentication. Each Savant has its own official public and private keys, to encrypt and decrypt messages, respectively. When two Savants communicate, they can use each other's public keys to encrypt the messages. Any listener without the private keys cannot understand the interaction. Therefore, if the responses received are appropriate, each Savant can believe the reported identity of the other. The most basic example of hardware verification via these means is for one Savant to send a message requesting the other to respond with the same message. If the response is accurate, the first machine knows that the second was able to decrypt the initial request.

The storage of these keys must be carefully chosen, since there is no point in having official keys if their source is untrusted or unprotected. Some sort of master server with all the keys for each piece of hardware is a possibility, similar to the certificate authorities currently used for authenticity on the Internet. Another approach is to distribute this data on physical media, such as CDs.

One important tool for protecting against various types of attacks is redundancy. By maintaining data on a number of different machines, the correct data can remain available when machines or connections are disabled. The CAN discovery service has provisions for

this, making sure that the nodes are responsible for overlapping portions of the data.

Protecting against software compromise is considerably more difficult, since it is more complex and much more easily modifiable. Since a newcomer Savant has a known piece of basic software, a cryptographic checksum can verify it. After setting up a custom configuration, however, the task becomes more complex because the correct checksum cannot be calculated beforehand.

The chance of system infiltration, even if all the Savants act in a genuine manner, is not negligible because every RFID reader is an access point to the system. Thus every Savant must be able to fend for itself against external attacks—no machine can be unconditionally trusted. This is a general precaution that should be taken for all network traffic.

Even though the network is already fairly open, it should not be directly addressable from the Internet, to prevent the possibility of various network attacks.

Chapter 8

Conclusion

In this thesis, I developed a system for Savant auto-configuration. Using NETCONF as the communication standard and CAN as the discovery service, a new Savant can obtain from other Savants its configuration data. The final solution involves a series of steps. First, the new Savant requests help from the local leader of its subnet to get its network configuration. Then, it compiles information about itself and its surroundings to be used to determine its function. The Savant then uses the discovery service to find its function, download software, and set up that software. Finally, it integrates with the other Savants, serving its purpose for the network.

In the solution, I chose a language, discovery service, and leader election algorithm to use. Although these are the best choices according to my analysis, there are certainly other possibilities. Many factors, such as network size, Savant hardware, and security needs, can affect these choices. For specific network implementations, further analysis would yield more information about the relative benefits of different choices.

While my solution uses discrete components, an alternative approach would be to develop a single, fully integrated solution from scratch. Although this would lack the benefit of code reuse, it could be more efficient than my solution, optimized to the specifics of Savant networks.

In my design, I have assumed that all Savants have access to enough resources to perform all the tasks necessary. A Savant network can be very heterogeneous, so this assumption may not always be true; some Savants may have too little processing power, too small storage capacity, or too little communication bandwidth, to give a few examples. Because new Savants typically compile information about their hardware as part of system configuration, my solution can potentially support a great variety of different Savant hardware. Still, it is possible for a Savant to have too few resources available to get to the system configuration step. In future work, my solution could be modified to make provisions for less capable Savants. For example, other Savants in the network could assist a new Savant to complete tasks it cannot do by itself. The fully integrated solution mentioned in the previous paragraph would have an advantage here, because it does not have to rely on the dependencies of discrete components.

The next step in developing my auto-configuration system is creating a simple implementation. Doing this would verify my design choices and identify ways to improve the system. I left a few decisions to the implementer because they depend on specific details of the network being configured. Implementation would lend further insight to the effects the network specifications have on these decisions.

Another important area for future study is security. In this thesis, I reviewed common attacks and countermeasures and their relation to Savant configuration. More study relating to the specific vulnerabilities of the Savant network and its configuration would yield valuable information about how best to secure my auto-configuration system.

Bibliography

[1] A. Bakker, A. S. Tanenbaum, E. Amade, G. Ballintijn, I. Kuz, I. Van Der Wijk, M. Van Steen, and P. Verkaik. The globe distribution network, March 28 2000.

[2] Jacob Brunekreef, Joost-Pieter Katoen, Ron Koymans, and Sjouke Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157–171, 1996.

[3] W. Bux. Token-ring local-area networks and their performance. *Proc. IEEE*, 77:238–256, 1989.

[4] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[5] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFC 3396.

[6] Daniel W. Engels, Ronald L. Rivest, Sanjay E. Sarma, and Stephen A. Weis. Security and privacy aspects of low-cost radio, August 25 2004.

[7] D. Estrin, R. Govindan, and J. Heidemann. Scalable coordination in sensor networks, 1999.

[8] David Karger, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Chord: A scalable peer-to-peer lookup service for internet applications, June 18 2001.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks, 2002.

[10] Oat Systems and MIT Auto-ID Center. *The Savant, Version 0.1 (Alpha)*, February 2002.

[11] Ed. R. Enns. NETCONF Configuration Protocol. Internet-Draft Version 6, IETF, April 2005.

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[13] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[14] Theodoros Salonidis, Pravin Bhagwat, Leandros Tassiulas, and Richard O. LaMaire. Distributed topology construction of bluetooth personal area networks. In *INFOCOM*, pages 1577–1586, 2001.

[15] Roy Want, Kenneth P. Fishkin, Anuj Gujar, and Beverly L. Harrison. Bridging physical and virtual worlds with electronic tags. In *CHI*, pages 370–377, 1999.

[16] Brandon Wiley, Ian Clarke, Oskar Sandberg, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system, February 10 2000.